

In the Claims:

Amend Claims 1, 3, 4, 8, 9 and 11-20 as follows:

Sum
OK
7
A 1. (Amended) A system comprising:
at least one thread;
a pool of locks;
at least one object that is capable of representing a resource needed by the at least one thread, the at least one object having a variable; and,
a recyclable locking mechanism for associating a lock from the pool of locks with the at least one object using the variable as a pointer when requested by the at least one thread, the lock returning to the pool of locks without having to destroy the at least one object when the at least one thread no longer needs to access the resource.

B1
OK 3. (Amended) The system of Claim 1, wherein the variable of each of the at least one object comprises an integer.

4. (Amended) The system of Claim 1, wherein the variable of each of the at least one object comprises a set of high bits defining the pointer to a lock and a set of low bits defining a status variable.

B1
OK 8. (Amended) The system of Claim 7, wherein upon the set of low bits after incrementation by one being greater than 0, the variable has an in-use status by a thread such that the set of high bits points to a lock.

9. (Amended) The system of Claim 7, wherein upon the variable after incrementation by one being less than 32, the associated variable has a spin status such that the set of high bits is currently in the process of being set to a lock.

LAW OFFICES OF
CHRISTENSEN O'CONNOR JOHNSON KINDNESS^{LLC}
1420 Fifth Avenue, Suite 2800
Seattle, Washington 98101
206.682.8100

11. (Amended) A method comprising:
asserting an instruction by a thread to lock an object;
increasing a variable of the object, the variable having a set of high bits for representing a pointer to a lock and a set of low bits for representing a lock status;
determining whether the variable is greater than a boundary value so as to allocate the lock; and
recycling the lock by returning the lock to a pool of locks when the thread no longer needs the object regardless of whether the object persists after the lock returns to the pool of locks.
12. (Amended) The method of Claim 11, further comprising initially setting the variable of the object to -1.
13. (Amended) The method of Claim 11, further comprising upon determining that the variable is less than the boundary value, waiting until the variable is greater than the boundary value.
14. (Amended) The method of Claim 11, further comprising upon determining that the variable is greater than the boundary value, using the set of high bits of the variable as a pointer to a lock for the object.
15. (Amended) The method of Claim 14, further comprising:
decrementing the variable of the object; and, determining whether the variable is less than a minimum threshold.
16. (Amended) The method of Claim 15, upon determining that the variable is less than the minimum threshold, recycling the lock.

17. (Amended) A computer comprising:
a processor;
a computer-readable medium; and,
a recyclable locking mechanism program executed by the processor from the medium to
associate a lock with an object using a variable of the object as a pointer when requested by a
thread, the lock being capable of returning to a pool of locks without having to destroy the object
when the object is no longer needed by the thread.

18. (Amended) The computer of Claim 17, wherein the variable of the object
comprises a set of high bits defining the pointer to a lock and a set of low bits defining a status
variable.

19. (Amended) A computer-readable medium having a recyclable locking
mechanism program stored thereon for execution on a computer to associate a lock with an
object using a variable of the object as a pointer when requested by a thread, the lock being
capable of returning to a pool of locks without having to destroy the object when the object is no
longer needed by the thread.

20. (Amended) The computer-readable medium of Claim 19, wherein the variable of
the object comprises a set of high bits defining the pointer to a lock and a set of low bits defining
a status variable.

REMARKS

The March 29, 2002, Office Action (the "Office Action") in this application rejected
Claims 1-3, 17, and 19 under 35 U.S.C. § 102(e) as being anticipated in view of the teachings of
U.S. Patent No. 6,237,043 issued to Brown et al. (hereinafter "Brown et al."). In addition,
Claims 4, 6-14, 18, and 20 were rejected under 35 U.S.C. § 103(a) as being unpatentable in view
of the teachings of Brown et al., taken in view of the teachings of U.S. Patent No. 6,098,089

LAW OFFICES OF
CHRISTENSEN O'CONNOR JOHNSON KINDNESS^{LLC}
1420 Fifth Avenue, Suite 2800
Seattle, Washington 98101
206.682.8100

issued to O'Connor et al. (hereinafter "O'Connor et al."). Furthermore, Claim 5 was rejected under 35 U.S.C. § 103(a) as being unpatentable in view of the teachings of Brown et al., taken in view of O'Connor et al., and U.S. Patent No. 5,687,073 issued to Kishimoto (hereinafter "Kishimoto"). Claims 15 and 16 were rejected under 35 U.S.C. § 103(a) as being unpatentable in view of the teachings of Brown et al., taken in view of the teachings of O'Connor et al., and U.S. Patent No. 5,797,004 issued to Lindholm et al. (hereinafter "Lindholm et al."). The drawings were objected to for lacking certain elements, which are referenced in applicant's specification. Portions of applicant's specification were objected to for various informalities. This amendment amends Claims 1, 3-4, 8, 9, 11, and 12-20 to clarify the subject matter of the claimed invention. In addition, minor amendments have been made to the specification, thereby rendering moot the objections set forth in the Office Action.

Prior to discussing in detail why applicant believes that all of the claims in this application are allowable, a brief description of applicant's invention and a brief description of the teachings of the cited and applied references are provided. The following background and the discussions of the disclosed embodiments of applicant's invention and the teachings in the cited and applied references are not provided to define the scope or interpretation of any of the claims of this application. Instead, such discussions are provided to help the United States Patent and Trademark Office (hereinafter "the Office") better appreciate important claim distinctions discussed thereafter.

Summary of Processes and Threads

Early operating systems allowed users to run only one program at a time. Users ran a program, waited for it to finish, and then ran another one. Modern operating systems allow users to execute (run) more than one program at a time or even multiple copies of the same program at the same time. This change highlights a subtle distinction: the difference between a program and a process. A program is a static sequence of instructions whereas a process is the

LAW OFFICES OF
CHRISTENSEN O'CONNOR JOHNSON KINDNESS^{PLLC}
1420 Fifth Avenue, Suite 2800
Seattle, Washington 98101
206.682.8100

dynamic invocation of a program along with the system resources required for the program to run. Thus, a process, in the simplest terms, is an executing program.

Processes include one or more threads. A thread is the basic unit used by the operating system to allocate processor time. A thread can include any part of the process code, including parts currently being executed by another thread. A processor is capable of executing only one thread at a time. However, a multitasking operating system, i.e., an operating system that allows users to run multiple programs, appears to execute multiple programs at the same time. In reality, a multitasking operating system continually alternates among programs, executing a thread from one program, then a thread from another program, etc. As each thread finishes its subtask, the processor is given another thread to execute. The extraordinary speed of the processor provides the illusion that all of the threads execute at the same time. Multitasking increases the amount of work a system accomplishes because most programs do not require that threads continuously execute. For example, periodically, a thread stops executing and waits while a slow resource completes a data transfer or while another thread is using a resource it needs. When one thread must wait, multitasking allows another thread to execute, thus taking advantage of processor cycles that would otherwise be wasted.

While the terms multitasking and multiprocessing are sometimes used interchangeably, they have different meanings. Multiprocessing requires multiple processors. If a machine has only one processor, the operating system can multitask, but not multiprocess. If a machine has multiple processors, the operating system can both multitask and multiprocess.

Summary of Synchronization

Threads often require a way to communicate with one another to coordinate their activities. While there are many complex forms of communication among threads, such as events, semaphores, timers, and mutants, the simplest form is called synchronization. Synchronization refers to the ability of one thread to voluntarily stop executing and wait until

LAW OFFICES OF
CHRISTENSEN O'CONNOR JOHNSON KINDNESS^{LLC}
1420 Fifth Avenue, Suite 2800
Seattle, Washington 98101
206.682.8100

another thread performs some operation. As briefly discussed in the example above, periodically, a thread stops executing and waits while a slow resource, such as a hard disk, completes a data transfer or while another thread is using a resource, such as a shared memory buffer, it needs.

As another example, consider a program with two threads, one with higher priority than the other, both threads requiring access to the same resource at some point. On a single processor system, the higher priority thread may have full access to the resource and will not have to relinquish control to the lower priority thread because the operating system gives preference to higher priority threads. However, on a multiprocessor system, both threads can run simultaneously, each on its own processor. This situation may cause each thread to simultaneously compete for access to the same resource, thereby creating the possibility of a race condition. A race condition exists when a thread modifies a resource to an invalid state, and then another thread attempts to access that resource and use it in the invalid state. One way to coordinate these two threads is through the use of synchronization.

One conventional synchronization technique is the use of a lock object. For example, a thread-safe multithreaded program typically is designed to have a resource data structure that represents a resource to be shared among threads in the program. This resource data structure can be synchronized by referencing to one of many synchronization capabilities provided by the environment in which the program will be run. Among the synchronization capabilities is the ability to create a lock object. The resource data structure also typically has a member function that accesses the resource to be shared. Whenever the member function of the resource data structure is invoked by a thread to access the resource, the member function creates a lock object and then calls the lock function of the lock object to prevent other threads from accessing the resource. When the member function finishes, the unlock function of the lock object is called to release the resource for other threads to use.

The approach of using lock objects as described above sacrifices efficiency for safety. The creation of one lock object may not take much time, but the creation of a multitude of lock objects may degrade computing performance to the point where most of the processor time is spent creating lock objects instead of doing tasks required by programs. These created lock objects also need to be maintained. Thus, not only must computing time be allocated to create these lock objects, but also to maintain them. Moreover, created lock objects consume resources (e.g., memory) that they do not even protect. Thus, a computing system becomes absurdly inefficient because a lock object exists for every object in the computing system, thereby doubling of the number of objects (and therefore spent precious computing resources) within the computing system just to enable objects to be locked.

Summary of the Invention

Applicant's invention is directed to avoid or reduce the unbounded creation of lock objects by a recycling process. More specifically, the system of the disclosed invention includes a pool of locks. The pool of locks has a set number of lock objects. By prescribing the use of lock objects from the pool of locks, the unbounded creation of lock objects is inhibited. One or more objects exist in the system of the disclosed invention, and at least one object represents a resource, such as a piece of data or a piece of hardware, that is needed by a number of threads. Each object that represents a resource has a variable, which can act to point to a lock in the pool of locks. The mechanism that causes the variable of an object to point to a lock, when a thread needs to access a resource represented by the object, is the recyclable locking mechanism of the disclosed invention. The lock returns to the pool of locks when the thread no longer needs to access the resource. The returning lock is in essence "recycled" for the use of another object, thereby conserving computing resources dedicated to creating or maintaining lock objects. Note that the lock is returned to the pool of locks whether or not the object continues to persist in the system of the disclosed invention.

LAW OFFICES OF
CHRISTENSEN O'CONNOR JOHNSON KINDNESS^{PLC}
1420 Fifth Avenue, Suite 2800
Seattle, Washington 98101
206.682.8100

Summary of Brown et al.

Like applicant's invention, the system of Brown et al. is directed to provide a locking mechanism, which is bound to an object by allowing the object to contain a pointer to the locking mechanism. But the similarity ends there. To understand the differences between the system of Brown et al. and the system of applicant's invention, it should be noted that the term "locking mechanism" as used by Brown et al. may be likened to a lock object of the applicant's invention but cannot be equated to the recyclable locking mechanism. The main problem is that the locking mechanism of Brown et al. remains bound to the object for the life of the object. See the Abstract of Brown et al.

As further emphasized at column 9, lines 52-54, Brown et al. indicates that "once a locking mechanism is bound to an object, it remains assigned to the object for the life of the object." Only when the object is destroyed (such as by garbage collection) will Brown et al. allow another object to use the locking mechanism entry in the locking mechanism table vacated by the destroyed object. In contrast, a lock of applicant's invention returns to the pool of locks whether or not the object that the lock previously secured is destroyed. In other words, a lock object of applicant's invention would return to the pool of locks because a thread no longer needs access to a resource being represented by an object—not because the object was destroyed.

The difficulty with the system of Brown et al. is that unless objects are destroyed, there can be no free locking mechanisms for other threads to use to synchronize access to shared resources. Consider the situation where no object can be destroyed. The system of Brown et al. would permanently stay the execution of multiple threads such that they will have to wait indefinitely to obtain a locking object (which these threads will never in fact obtain) to access shared resources. Not only would this create inadvertent deadlocks, but to solve this problem, Brown et al. would need to provide far more locking objects than can be anticipated. This would be a waste of computing resources.

Summary of O'Connor et al.

The system of O'Connor et al. is directed to implementing garbage collectors. Garbage collectors are processes that automatically recover unused memory of a computing system. More specifically, the system of O'Connor et al. is a framework for generation isolation by trapping intergenerational pointer stores. Identification of pointer stores as intergenerational is performed by a store barrier, which is responsive to a trap table. The trap table is programmatically encoded with store target object and store pointer data generation pairs to be trapped. The store barrier and the trap table allows a garbage collector to be implemented with support for a wide variety of generational garbage collection methods.

As discussed above, garbage collection is a way of destroying an object, which is required by the system of Brown et al. to free up locking mechanisms. O'Connor et al. fails to teach anything that cures the defects of Brown et al. Consequently, there is no benefit to combining Brown et al. and O'Connor et al., if the combination were even possible, which applicant specifically denies. For example, O'Connor et al. fails to teach the ability of recycling a lock by returning the lock to a pool of locks when a thread no longer needs an object that was secured by the lock. Even if Brown et al. and O'Connor et al. were combinable, the resulting combination would not anticipate applicant's invention.

The Office has cited O'Connor et al. for the further proposition that it teaches a 32-bit variable for storing both a pointer and a status. Applicant respectfully disagrees. The system of O'Connor includes a "32-bit word that itself includes a method vector table base portion 612 for representing object's class and five bits of additional storage 614 reserved for synchronization status of the object and information for the garbage collector." *Id.* at column 13, lines 54-58. Thus, the five bits must be used for both synchronization status as well as for information for garbage collection. In order to point to a monitor, an additional 32-bit word must be allocated. *Id.* at column 13, lines 59-60 ("a second header-word, e.g., monitor pointer 616 can contain the

addresss of a monitor allocated for the object"). In other words, 64-bit words must be used in the system of O'Connor et al. instead of a 32-bit word.

The system of O'Connor et al. requires monitors, but Brown et al. decries the use of monitors: "While the use of monitors prevents race conditions from occurring, this approach can significantly degrade the performance of the information handling system." *Id.* at column 3, lines 10-12. To combine, either the approach of Brown et al. must be abandoned or the approach of O'Connor et al. must be jettisoned, and the combination would destroy the operation of either reference. Even if the combination of Brown et al. and O'Connor et al. were possible, which applicant specifically denies, these references cannot anticipate or render applicant's invention obvious.

Summary of Kishimoto

The system of Kishimoto is directed to an Initial Program Loading (IPL) System for a multiprocessor system that has a plurality of processors. Each processor is connected to another processor through a communication path. The IPL system of Kishimoto prevents any one of the processors from suffering from an excessive load that may be exacted by the IPL process when it is executing. Furthermore, the IPL system of Kishimoto shortens the time needed to carry out the IPL process.

The system of Kishimoto has nothing to do with providing recyclable locks. Consequently, there is no benefit to combine Brown et al., O'Connor et al., and Kishimoto if the combination were even possible, which applicant specifically denies. Neither Brown et al. nor O'Connor et al. teach the ability of recycling a lock by returning the lock to a pool of locks when a thread no longer needs an object that was secured by the lock. Even if Brown et al., O'Connor et al., and Kishimoto were combinable, the resulting combination would not anticipate applicant's invention.

Summary of Lindholm et al.

The system of Lindholm et al. is directed to caching and allocating thread synchronization constructs. To synchronize between a plurality of threads of execution and a set of objects, the system of Lindholm et al. provides an object synchronization module, which comprises a cache of synchronization constructs, a hash table containing a list of pointers pointing to allocated synchronization constructs, a list of free synchronization constructs, and a cache manager. When a requesting thread seeks synchronization with an object, the cache manager allocates a synchronization construct in the list for synchronizing the requesting thread with the object only when there is a free synchronization construct in the list. During the allocation of the synchronization construct, a pointer in the list of pointers is caused to point to the allocated synchronization construct. Moreover, each synchronization construct contains an object identifier, which holds the address of the object being synchronized by the corresponding synchronization construct. If there are no free synchronization constructs (in other words, all of them have been allocated for synchronization among other threads and objects), the requesting thread is placed in a waiting list of the requested synchronization construct. The cache manager de-allocates a synchronization construct and returns the synchronization construct to the free list by causing the pointer that points to the de-allocated synchronization construct to point to the free list.

Unlike applicant's invention, objects in the system of Lindholm et al. lack a variable that can be used as a pointer to point to a lock object and/or be used to indicate the lock status. This minimal design avoids the complexity and the waste of computing resources associated with the use of the object synchronization module of the system of Lindholm et al. It should be noted, however, that each pointer in the list of pointers of the system of Lindholm et al. exists in the context of the object synchronization module to point to a synchronization construct, and therefore, these pointers cannot be likened to the variables of objects of applicant's invention. In

fact, objects in the system of Lindholm et al. do not point to allocated synchronization constructs but instead allocated synchronization constructs point to corresponding objects. To accomplish this, the system of Lindholm et al. requires significant overhead, such as a cache of synchronization constructs, a cache manager, a hash table, a hash table address generator, a free list, a cache size controller, and a hash table size controller, in order to manage synchronization constructs so that they can point to corresponding objects. Not so with applicant's invention because synchronization is focused on a simple variable existing in an object that represents a shared resource.

The system of Lindholm et al. can be used to synchronize an object using three approaches, but not one of them uses a variable in an object for synchronization. *See* Lindholm et al. at column 3, line 63, to column 4, line 33. The first of these three approaches consists of the use of a synchronized method of an object. When a synchronized method of an object is executed by a thread, the object that contains the synchronized method will be synchronized with and owned by that thread until the execution of the method is terminated. The second approach is the use of critical sections within a method of an object to cause synchronization with a thread when the section protected by a corresponding critical section is executed by the thread. The final approach is the use of a method of an object that contains a call to another object or a method of another object, which is declared to be synchronized. In summary, the system of Lindholm et al. uses specialized methods to synchronize whereas applicant's invention synchronizes objects by the use of simple variables. Methods are more complicated to maintain than variables. Moreover, methods require greater computing resources than variables. No one skilled in the art would choose a more expensive and less efficient technique to synchronize objects. Thus, the technical distinction between method members of an object and a variable member of the object should be kept distinct to understand applicant's invention.

In the Field of the Invention, the system of Lindholm et al. is described as requiring "an object synchronization module and associated method that uses a cache of monitors for synchronizing objects." Brown et al. criticizes the use of a cache of monitors (monitor pool): "While the use of monitors prevents race conditions from occurring, this approach can significantly degrade the performance of the information handling system." *Id.* at column 3, lines 10-12. Furthermore, "once a locking mechanism is bound to an object, it remains assigned to the object for the life of the object." *See* Brown et al. at column 9, lines 52-54. To combine, either the approach of Brown et al. must be abandoned or the approach of Lindholm et al. must be jettisoned, and the combination would destroy the operation of either reference. Even if the combination of Brown et al., O'Connor et al., and Lindholm et al. were possible, which applicant specifically denies, these references cannot anticipate or render applicant's invention obvious.

The Claims Distinguished

The Office has failed to show, and applicant is unable to find, where any of the cited and applied references, either alone or in combination, disclose the subject matter of the claimed invention. Among other differences, none of the applied and cited references teaches "a recyclable locking mechanism for associating a lock from the pool of locks with the at least one object using the variable as a pointer when requested by the at least one thread, the lock returning to the pool of locks without having to destroy the at least one object when the at least one thread no longer needs to access the resource" as recited in amended independent Claim 1. Similar language is now included in amended independent Claims 11, 17, and 19. Brown et al. requires the object to be destroyed through a destruction process, such as a garbage collector, before the locking mechanism can be used by another thread. O'Connor et al. provides the garbage collector to destroy objects. Kishimoto teaches nothing about this recitation among other things. Lindholm et al. fails to teach the use of a variable of an object as a pointer to a lock. In short, these cited and applied references do not teach or even remotely suggest the claimed invention.

LAW OFFICES OF
CHRISTENSEN O'CONNOR JOHNSON KINDNESS^{LLC}
1420 Fifth Avenue, Suite 2800
Seattle, Washington 98101
206.682.8100